

Towards Verification of Constituent Systems through Automated Proof

Luís Diogo Couto

Aarhus University, Denmark
ldc@eng.au.dk

Simon Foster

University of York, United Kingdom
simon.foster@york.ac.uk

Richard Payne

Newcastle University, United Kingdom
richard.payne@ncl.ac.uk

Abstract—This paper explores verification of constituent systems within the context of the *Symphony* tool platform for Systems of Systems (SoS). Our SoS modelling language, CML, supports various contractual specification elements, such as state invariants and operation preconditions, which can be used to specify contractual obligations on the constituent systems of a SoS. To support verification of these obligations we have developed a proof obligation generator and theorem prover plugin for Symphony. The latter uses the *Isabelle/HOL* theorem prover to automatically discharge the proof obligations arising from a CML model. Our hope is that the resulting proofs can then be used to formally verify the conformance of each constituent system, which in turn would result in a dependable SoS.

I. INTRODUCTION

A System of Systems (SoS) [1] is a collection of semantically heterogeneous, independent, and distributed constituent systems (CSs) which are co-ordinated to achieve an overall goal. Independence means that no CS can exert control on another CS, only influence its behaviour by offering potential opportunities should synergy be reached. Since CSs are dynamic and heterogeneous, often changing their capabilities and services, such synergy is achieved by negotiation of *contracts* between a set of CSs, which impose binding conditions on the behaviour of each CS. Since failure of such an agreement will result in degradation of the SoS, it is important that each CS has some measure of certainty in its ability to fulfil its requirements, which in turn will lead to a dependable SoS.

System of Systems Engineering (SoSE) therefore requires languages with which we can accurately model CSs to predict their behaviour, and tools which enable their verification. Such languages should have a sound theoretical background to ensure that they can be assigned a consistent behaviour, and the ability to handle the composition of heterogeneous constituents. To this end the COMPASS Modelling Language (CML) [2] has been developed, a formal modelling language for SoSs. CML reproduces the style of the VDM-SL [3] formal specification language, whilst integrating CSP [4] process modelling constructs from the *Circus* [5] language. CML has a formal semantics based in Hoare and He's Unifying Theories of Programming (UTP) [6], in its denotational, operational, and axiomatic flavours. Along with the associated *Symphony*¹ tool platform, CML allows SoSs and CSs to be formally modelled, tested, and verified in a controlled environment.

This paper focuses on two closely related components of Symphony, the Theorem Prover Plugin (TPP) and Proof

Obligation Generator (POG). A theorem prover can be applied to *verify* a software system, that is mathematically demonstrate that required properties are met through mechanically verified proof. In the case of CSs, we need to verify that the internal functionality and pattern of interaction is guaranteed to fulfil the contract. In Symphony this verification can be facilitated through the POG which generates proof goals upon which the correctness of the CS model depends. CML has a number of facilities for specifying contractual obligations, such as type invariants, pre- and post-conditions for functions and operations, and system state invariants. These can then variously be used to specify contractual obligations for a CS model, and the application of the POG in concert with the TPP can be used to verify that the system satisfies those obligations. Our thesis is, therefore, that these technologies provide a way forward for mechanically verifying that a CS model fulfils its contractual obligations to the wider SoS.

In the remainder we outline our contributions. Section II gives more background to our baseline technology. Section III discusses related work. Section IV discusses the combined POG and TPP framework, and how it can be used to verify a CS model. Section V demonstrates an example CS, and how we envisage verifying it for a wider SoS. Finally in Section VI we conclude and outline future work.

II. BACKGROUND

CML is a language for modelling constituent systems and their composition in an SoS. Systems are modelled using CML *processes*, which are stateful reactive entities that can be executed concurrently, and exchange messages over *channels* in the style of the CSP process calculus [4]. A CML model consists of a collection of user defined *types*, *functions*, *channels*, and *processes*. A process, in turn, consists of private *state variables*, *operations* that act on these variables, and *actions* that specify reactive behaviour using operators from CSP. CML processes can be parallel composed to represent concurrent execution, enabling description of a complete SoS.

CML has a formal mathematical foundation [7] based in the UTP semantic framework [6], which allows processes to be given a precise semantics. UTP allows us to tackle semantic heterogeneity in SoSE by decomposing a modelling language semantics into its theoretical building blocks, such as state, concurrency, discrete time, and mobility, which can then be formalised as “*UTP theories*”. UTP theories then act as components with which we can construct semantic models for languages and provide links between similar languages based on common theoretical factors.

¹Symphony can be downloaded from <http://symphonytool.org/>

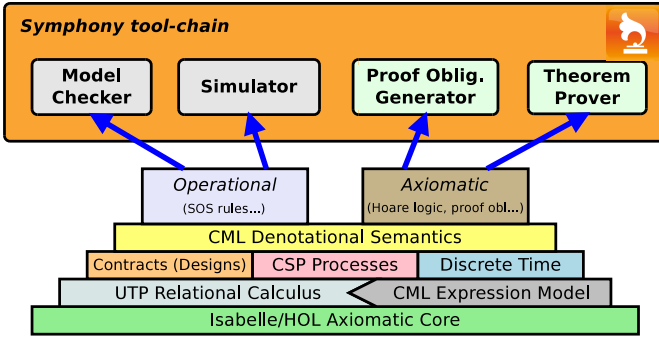


Fig. 1: Semantically supported Symphony tool platform

Development of CML models is aided through the associated *Symphony* tool platform. Symphony is an Eclipse-based development environment that provides a parser, syntax highlighting, a type checker, simulator, model checker, and a variety of other tools for variously constructing and verifying CML models. Though consisting of independently developed components, the different tools share a common semantic foundation given by the UTP denotational model. This “semantic stack” is shown in Figure 1, with the Symphony platform and associated components positioned on top. Within the UTP relational calculus several computational paradigms have been formalised as UTP theories (Contracts, Processes, etc.), and these have been in turn composed to produce the CML denotational semantics. Finally, reference semantics have been produced that underlie the various tools, including the operational semantics, which underlies the simulator and model checker, and various axiomatic semantics, such as a Hoare calculus [8]. Since this formal link exists from each tool down into the unified semantic basis we can have a degree of certainty that the various evidences produced can be consistently composed to verify a CS model.

To support such verifications we have created a theorem prover plugin, based on the *Isabelle/HOL* [9] interactive theorem prover. Isabelle/HOL is ideal for this kind of verification since proofs can be independently checked with respect to a secure axiomatic core; a facet of the “*LCF architecture*”. Our theorem prover is based in a mechanised semantic framework for UTP called *Isabelle/UTP* [10] that provides a strong theoretical grounding for CML, ensuring its consistency. We have mechanised a partial semantic model for CML in *Isabelle/UTP*, a collection of associated proof tactics, and a visitor that translates the CML Abstract Syntax Tree (AST) into Isabelle definitions that can then be used to support proof. Our current approach to proof in CML is to, where possible, convert CML to equivalent HOL formulae, and perform the proof using Isabelle’s variety of existing tactics and laws, effectively transferring results from HOL to UTP. In line with the UTP framework, the theorem prover is fully *extensible*: we can add support for additional programming concepts, and associated tactics as required in the future.

Alongside the theorem prover, Symphony contains a POG that generates, for a given CML document, a collection of proof goals that must be satisfied to prove certain high level properties of the model, such as internal consistency, contractual correctness of operations, and termination. The TPP can then be used to attempt discharge of these proof obligations, resulting in concrete proof objects for the properties. We are

currently working towards a formal axiomatic semantics for these proof obligations based on the current implementation, which will allow the integration of these proof objects with other evidences in the tool-chain.

III. RELATED WORK

The Symphony tool platform is an extension of the open source *Overture* IDE [11] for VDM based modelling. The Symphony POG is an adaptation of the POG for *Overture* [12] to also handle CML proof obligations. Previous efforts to generate and discharge Proof Obligations (POs) for VDM include [13] and [14], which connect VDMTools and *Overture* POs respectively to the HOL4 theorem prover [15]. These attempts were limited to the functional subset of VDM. We use a similar mapping for CML types and expressions, whilst adding support for CML’s imperative and concurrent constructs.

The area of theorem proving tools includes a number of options including Isabelle/HOL [9] (which we use); PVS² combining a specification language with a theorem prover; Coq [16], a proof assistant based on intuitionistic logic; specialised verification systems such as *Spec#*, which is based on the Boogie verification language [17] and supported by the Z3 SMT solver [18]; and the *Rodin*³ tool for Event-B which includes an automated theorem prover.

We choose Isabelle for several reasons. It is based on *Higher Order Logic* which is ideal for embedding a language like CML. The LCF architecture ensures proofs are correct with respect to a secure logical core. It has a large library of mathematical structures related to program verification, such as relational calculus and lattice theory. It integrates powerful proof facilities, such as the *auto* tactic for automated deduction, integration of first-order automated theorem provers (like Z3) in the *sledgehammer* tool [19], and counterexample generators like *nitpick*. We can directly harness many of these proof facilities by our transfer based proof tactics. Finally, Isabelle has been integrated into Eclipse in the form of *Isabelle/Eclipse*⁴, an IDE which we reuse in Symphony.

IV. PO GENERATION AND DISCHARGE IN SYMPHONY

The Symphony POG is an extension of the *Overture* POG for the Vienna Development Method (VDM) [12], and therefore many proof goals generated are derived from VDM. However, the POG has been developed with an extensible visitor [20] based architecture that will enable the addition of further goals as they are researched. The current proof goals fall broadly into the following categories: safe usage of partial operators; safe usage of functions with pre-conditions; type compatibility due to union types, type invariants and subtypes; and satisfiability of implicitly defined functions and operations.

To illustrate the use of POs in Symphony, we present a simple example based on a well known partial operator case: division by zero. Consider the following CML function:

```
division : int * int → real
division (x,y) == x / y
```

²<http://pvs.cdl.sri.com>

³<http://event-b.org>

⁴<http://andriusvelykis.github.io/isabelle-eclipse/>

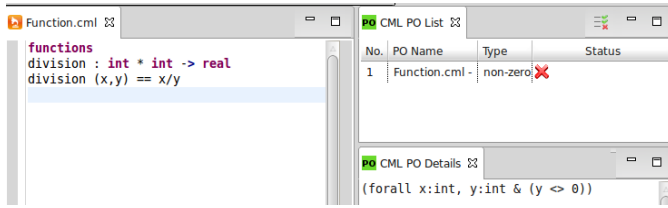


Fig. 2: Failed PO discharge.

For this function, the POG generates an obligation stating that, for all inputs to the function, the value of the divisor (y) will not be 0, thus ensuring the function executes successfully. This is represented in the logical formula below:

```
PO1: forall x:int, y:int & (y <> 0)
```

This states that for all variables x and y of type `int`, y is not equal to 0. This is not satisfiable, and so an attempt to discharge this PO will fail as shown in Figure 2. The function must be enriched with a pre-condition, using the **pre** keyword, in order for the discharge to be possible:

```
division : int * int -> real
division (x,y) == x / y
pre y <> 0
```

The additional information offered by the pre-condition alters the PO and now the theorem prover plug-in is able to discharge the revised PO as shown in Figure 3. It is of course not possible to prove that an arbitrary integer is different from zero, but it is trivial to prove that a non-zero integer is different from zero.

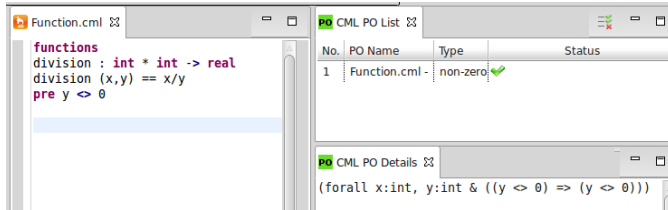


Fig. 3: Successful PO discharge.

The aforementioned example was trivial but, in general, it is quite important to ensure that, when adding a pre-condition, said pre-condition is sufficient to allow the discharge of any POs generated for the function.

The addition of the pre-condition has another effect. One must now ensure that, whenever the function is called its pre-condition is respected. Therefore, a new kind of PO is generated. Consider the following function and PO:

```
divby2: int -> real
divby2 (x) == division(x,2)
```

```
PO2: forall x : int & pre_division(x,2)
```

Since `divby2` calls `division`, a PO is generated to ensure that the pre-condition of `division`, given by `pre_division`, is satisfied. This kind of obligation, called pre-condition obligation is generated at all points in the model where the function is called.

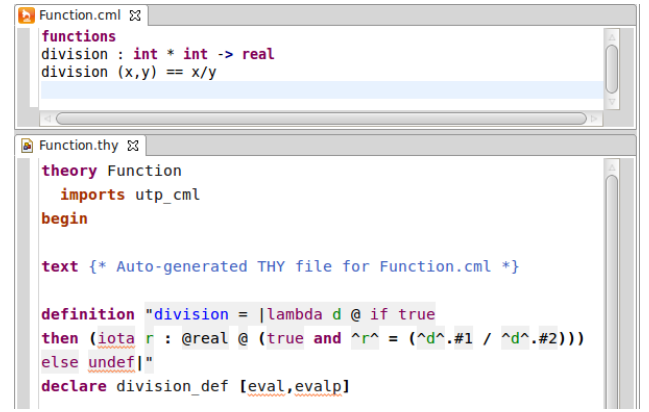


Fig. 4: Auto-generated theory files.

While the example shown was very simple, pre-conditions (and invariants) can be as complex as necessary. They are expressed in the functional subset CML and thus have the full expressive power of CML's first-order logic. In addition to helping ensure consistency of the model, pre-conditions and invariants are also used to specify additional properties

In fact, the methodology we propose is based precisely on specifying desired properties and requirements of a model through pre- and post-conditions as well as invariants. The POs, once generated and discharged, stand as proof that the model respects the specified properties.

Discharging POs is the task of the TPP. At its core, the TPP consists of a mechanised semantic model for CML within *Isabelle/UTP*. It is essentially a deep embedding of CML, in that we give an explicit semantics to each of the operators of CML processes within *Isabelle*.

The TPP will process a CML model and its associated POs and automatically generate *Isabelle* theory files for them (see Figure 4). These theory files can then be submitted to *Isabelle* for discharging through various automated proof tactics such as `auto` and `sledgehammer`, or the `cml_tac` tactic that maps a CML formula onto a HOL formula.

Because the TPP connects to the *Isabelle/Eclipse* plug-in, the full functionality of that plug-in and, by extension, *Isabelle* is available to the user. This includes the ability to write and discharge model-specific conjectures directly in the *Isabelle* encoding of the model. However, to perform this kind of work requires significant knowledge of *Isabelle* and its syntax.

Therefore, the POG will be the primary source of goals to discharge. Furthermore, the TPP offers a fully automated mode of interaction with *Isabelle* where users simply choose which PO to discharge and all inner workings (such as tactic selection and result collection) are hidden from them.

We envisage two main functionalities for the plug-ins *Quick check* and *Proof Session*. *Quick check* will be a fully automated process, simply presenting a list of Proof Obligations (POs) (including their predicates) in CML and linked to the relevant model elements. The process of generating POs is quick, therefore this may be performed frequently during initial model development, to gain useful feedback about the model.

The *Proof Session* will be the main functionality of the plug-ins. It creates a snapshot of the model (a timestamped,

read-only copy of the model's CML sources), generates POs and translates the model and POs into Isabelle theories. The POs are displayed in a similar manner to the quick check version but can now be submitted to the TPP for discharge. At the moment only `cml_tac` is available, though we hope to enable automated use of additional tactics when attempting to discharge POs. Regardless, the output of a proof attempt will be captured from Isabelle and displayed to the user. Also, the results of a proof session will be stored along with the model snapshot, thus verifying the model's correctness.

These two functionalities combine to form the following work-flow: as a user works on a model, he can quick check for POs as a way to gain early insights into the of the model. Each PO can be seen of as a possible inconsistency and merely by manual inspection they can guide the user in terms of adding necessary pre-conditions or guards to the model.

Once a set of changes has been completed, the user may use the proof session functionality to verify the model's correctness. Each set of POs and their associated proofs are only valid for the particular version of the model they were generated from so it makes little sense to attempt manual proofs on a volatile model. Regardless of when it is attempted, the proof session for the average user will be fully automated. The user simply initiates a proof session and selects POs for discharging either manually or in batch. Typically some POs will be successfully discharged whereas others will fail to discharge. These should indicate a problem with the model and action must be taken by the user (for example, by adding a guard or correcting program logic) to alter the model in a way that allows the PO to be discharged. Then, the set of completed PO goals can be used as a formal proof of the constituent system's correctness.

For advanced users who are comfortable interacting with *Isabelle/Eclipse* directly, the full theorem proving perspective gives them direct access to the tool so that manual proofs may be attempted. Users can also specify and discharge additional model-specific conjectures.

V. VERIFICATION OF EXAMPLE CONSTITUENT SYSTEM

We illustrate the use of the Symphony tool platform POG and TPP with a simple example CS from a Railway Signal System of Systems (SoS). The SoS in question aims to ensure the safe and correct movement of trains on a section of railway track. Naturally such a SoS poses several dependability concerns and the integrator of the SoS requires several safety properties to hold throughout the life of each of the systems.

The *Railway Signal* SoS comprises several constituents including a *Route Rule Engine*, several *Track Actuators*, *Trains* and *Dwarf Signal* systems. In this paper, we look at one of the constituent systems – the *Dwarf Signal* system – in detail and consider the safety properties of that system.

From the perspective of the SoS integrator, there is a requirement that the procured constituent systems provide a safe service. The constituent system designer must, therefore, provide evidence of this safety. Using model-based techniques, we define a formal model of the *Dwarf Signal* – which may be used as a contract to which the the signals must conform. The *Dwarf* model used in this paper is based upon that introduced in [21], and a typical signal may be seen in Figure 5.



Fig. 5: Picture of railway signal, with lamps indicated

The *Dwarf Signal* model is defined in CML with several datatypes, functions, a single *Dwarf* process with state variables, operations and actions. The main datatype, *DwarfType* shown below, has several fields relating to the transitions which are to be made in the *Dwarf Signal*. For example, the *currentstate* field dictates the collection of lamps currently lit, and the *desiredproperstate* field represents the next state the *Dwarf Signal* should reach. The set of possible signal states that may be reached is defined by the *ProperState* datatype, which is constrained to be one of for constant values: *dark*, *stop*, *warning* and *drive* – each a set of lamps.

```
types
  LampId      = <L1> | <L2> | <L3>
  Signal      = set of LampId
  ProperState = Signal
  inv ps == ps in set {dark, stop, warning, drive}

  DwarfType :: lastproperstate : ProperState
              turnoff          : set of LampId
              turnon           : set of LampId
              laststate         : Signal
              currentstate      : Signal
              desiredproperstate : ProperState
  inv d == NeverShowAll(d) and MaxOneLampChange(d)
           and ForbidStopToDrive(d) and DarkOnlyToStop(d)
           and DarkOnlyFromStop(d)

values
  dark: Signal = {}
  stop: Signal = {<L1>, <L2>}
  warning: Signal = {<L1>, <L3>}
  drive: Signal = {<L2>, <L3>}
```

There are several safety properties to which the *Dwarf Signal* must adhere. These are defined in terms of functions referred to in the *DwarfType* invariant – including, for example, *NeverShowAll* which requires that the *currentstate* should never have all three lamps lit. The *Dwarf* process, outlined below has a single state variable: *dw* of type *DwarfType*, and four operations: *Init*, which initialises the *dw* state variable; *SetNewProperState*, allowing the next desired *properstate* to be set; and two operations for changing the lamps lit in the signal – *TurnOn* and *TurnOff*.

```

process Dwarf = begin
state
  dw : DwarfType

operations
  Init : () ==> ()
  Init() == (...)

  SetNewProperState: (ProperState) ==> ()
  SetNewProperState(st) == (...)

  TurnOn: (LampId) ==> ()
  TurnOn(l) == (...)

  TurnOff : (LampId) ==> ()
  TurnOff(l) == (...)
... end

```

Each operation is defined in more detail in terms of pre- and post-conditions, dictating the conditions in which the operation may be called and the guarantees it makes if those conditions are met. The *Init* operation, defined in more detail below, has a body which initialises the *dw* state variable, with a post-condition requiring that various fields of the *dw* variable are updated. The operation body – an assignment to the *dw* state variable – must respect the safety properties of the *Dwarf Signal*, in the form of the type invariant described above.

```

Init : () ==> ()
Init() ==
  dw := mk_DwarfType(stop, {}, {}, stop, stop, stop)
post dw.lastproperstate = stop and dw.turnoff = {}
    and dw.turnon = {} and dw.laststate = stop
    and dw.currentstate = stop
    and dw.desiredproperstate = stop

```

The remainder of the CML operations are defined in a similar manner – with pre- and post- conditions. In addition to these operation definitions, the CML model contains *actions* which dictate the ordering of internal events and operation calls. At present, the POG does not handle these features of CML, and thus they are omitted from this paper.

Executing the Symphony POG, we obtain several POs, which are generated by the *Init*, *SetNewProperState*, *TurnOn* and *TurnOff* operations. The POs fall into two PO types: ensuring that the postcondition holds given the body of the operation; and ensuring subtype consistency. It is the second of these which ensures that the *DwarfType* type invariant (and thus the safety properties of the *Dwarf Signal*) holds when setting a new value of the *dw* variable. The generated subtype POs (PO1 and PO2) and postcondition PO (PO3) for the *Init* operation are shown below.

```

PO1: inv_ProperState(stop)

PO2: ((inv_DwarfType(mk_DwarfType(stop, {}, {}, stop, stop, stop)) and inv_ProperState(stop)) and inv_ProperState(stop))

PO3: (((dw.lastproperstate) = stop) and (((dw.turnoff) = {}) and (((dw.turnon) = {}) and (((dw.laststate) = stop) and (((dw.currentstate) = stop) and (((dw.desiredproperstate) = stop))))))

```

Using the Symphony tool platform, we generate these POs, and attempt to discharge them. In Figure 6 below, we show

Symphony in the POG perspective with the POs represented in the Isabelle syntax used by the TPP. In the figure, the list of POs is given in the right hand pane, with a pane showing the PO definition in CML below. In the figure, we see that several of the POs have been discharged – these relate to the *Init* operation above – as denoted by the green ticks.

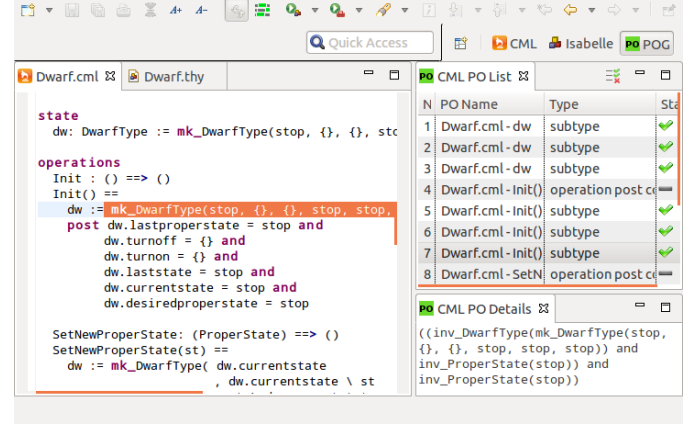


Fig. 6: Progress on POs generated for Dwarf model

By discharging all POs for the *Dwarf Signal* model, we provide a contractual model which is verified to be both internally consistent and, through encoding the safety properties which must be met by a signal, is safe with respect to the requirements placed on that contract. At present, whilst those POs shown are successfully discharged by the Symphony TPP, several are not. These relate to those POs which rely upon the value of the *Dwarf* process state variable *dw* at a given point of time. This may be either an issue with the PO expressions themselves (where the VDM-based PO expressions require further adaption to CML), or due to the early stage of development of the TPP proof tactics. We discuss these areas as future work in the next section.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have outlined two Symphony tool platform plugins which enable automated proof support for CML POs. The Symphony POG reuses and expands upon the Overture POG, also resulting in improvements in the Overture Tool. The TPP is the first attempt at tightly integrating a theorem prover into the VDM family of tools, providing a more useful tool for users wishing to discharge CML POs and general theorems. We have also shown how both plug-ins can be used in combination to formally verify the integrity of constituent systems of an SoS specified with CML. There are clearly many areas of future work, both short-term improvements to the two plugins, and also longer-term directions and scoping of the work in the fields of SoS and dependability-related issues. Below, we discuss several such directions for further work.

This paper demonstrated the verification of constituent system *models*. An interesting issue would be the verification of an actual implementation, with respect to realistic system properties. Whilst clearly not in the scope of this paper, we would consider the work of this paper in context with other system engineering activities. In particular; positioning constituent system verification with respect to the work of Holt et al [22] on SoS requirements engineering and the

specification of SysML contracts and translation to CML [23] may provide the means to more realistic property verification.

The current CML TPP focuses on VDM-style proof obligations which deal with issues such as subtyping and internal consistency. In the future we will extend this with a more comprehensive calculus, such as a Hoare logic [8] or a weakest precondition calculus, which would both expand on the existing proof obligations. This would also allow us to reason directly about CML process and state behaviour, and therefore provide fuller support for reasoning about contracts.

Another extension to the proof obligations relates to scaling our approach from the level of constituent systems to the SoS-level, thus ensuring that the verified constituent systems interact in a manner that ensures the desired behaviour of the overall SoS.

Just as pre-conditions and invariants can be used to specify the properties that the POG and TPP verify, we need a mechanism that allows these tools to reason about correctness at the SoS level. We see two distinct possibilities here: the first is to introduce a new CML construct that allows one to specify invariants over the entire SoS, thus being able to “see” inside all constituents. The second approach is to take the existing POs that verifies a system and use them to also verify the interface of a constituent. Afterwards, one must establish a means by which these verified interfaces can be combined to establish global SoS properties. Of the two approaches, the second one seems closer to the spirit of SoS engineering, and we believe CS refinement provides a way forward here.

We are also currently working on a tool for CS refinement, which combines with the theorem prover and can be used to formally demonstrate contractual satisfaction. This will reuse the POG to enumerate and discharge refinement provisos which must often be satisfied to ensure validity of a refinement step. Refinement will be principally supported by Isabelle, though we are also exploring the use of model generation tools to aid automation. For example, the Maude rewriting logic engine [24] has previously been applied to automated refinement [25], which we hope to adapt for CML. Such advances over the current technology are feasible because of our extensible approach to semantics provided by UTP.

Finally, though both plug-ins presented here are still at an early development stage, work is ongoing on various improvements. While the TPP and its associated Isabelle theories support a significant subset of CML (types, expressions, functions, and operations), work is ongoing on increasing the coverage of the plug-in. The proof tactics are also under further development in order to discharge increasingly complex goals. Moreover we wish to expose more of Isabelle’s native proof facilities in the TPP, such as *sledgehammer* and *nitpick*, so as to bring their full weight to bear in discharging or refuting proof obligations. Parallel to this there is work to formalise the proof obligations in Isabelle with respect to the CML semantics. Finally, we hope to produce guidance to the user of how to interpret failure when a PO cannot be discharged.

ACKNOWLEDGEMENTS

This work is supported by EU Framework 7 Integrated Project “Comprehensive Modelling for Advanced Systems of

Systems” (COMPASS, Grant Agreement 287829). For more information see <http://www.compass-research.eu>.

REFERENCES

- [1] H. Kopetz, “System-of-Systems Complexity,” in *Proc. 1st Workshop on Advances in Systems of Systems*, 2013, pp. 35–39.
- [2] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, “Features of CML: a Formal Modelling Language for Systems of Systems,” in *Proc. 7th Intl. Conference on Systems of Systems Engineering (SoSE)*. IEEE, July 2012.
- [3] C. B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 1990.
- [4] T. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [5] J. C. P. Woodcock and A. L. C. Cavalcanti, “A Concurrent Language for Refinement,” in *IWFM’01: 5th Irish Workshop in Formal Methods*, ser. BCS Electronic Workshops in Computing, July 2001.
- [6] T. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall, 1998.
- [7] J. Fitzgerald, P. G. Larsen, and J. Woodcock, “Foundations for Model-based Engineering of Systems of Systems,” in *Complex Systems Design and Management*, M. A. et al., Ed. Springer, January 2014, pp. 1–19.
- [8] S. Canham and J. Woodcock, “CML Definition 3 — Hoare Logic,” COMPASS Deliverable, D23.4d, Tech. Rep., September 2013.
- [9] T. Nipkow, M. Wenzel, and L. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.
- [10] S. Foster, F. Zeyda, and J. Woodcock, “Isabelle/UTP: A mechanised theory engineering framework,” in *5th Intl. Symposium on Unifying Theories of Programming*, 2014.
- [11] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, “The Overture Initiative – Integrating Tools for VDM,” *SIGSOFT Software Engineering Notes*, vol. 35, no. 1, Jan 2010.
- [12] L. Couto and R. Payne, “The COMPASS Proof Obligation Generator: A test case of Overture Extensibility,” in *Proc. 11th Overture Workshop*, 2013.
- [13] S. Agerholm and K. Sunesen, “Reasoning about VDM-SL Proof Obligations in HOL,” IFAD, Tech. Rep., 1999.
- [14] S. Vermolen, “Automatically Discharging VDM Proof Obligations using HOL,” Master’s thesis, Radboud University Nijmegen, Computer Science Dept., August 2007.
- [15] K. Slind and M. Norrish, “A brief overview of HOL4,” in *TPHOLS*, ser. LNCS, vol. 5170. Springer, 2008, pp. 28–32.
- [16] Y. Bertot and P. Castéran, *Coq’Art: the calculus of inductive constructions*. Springer, 2004.
- [17] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” in *Formal Methods for Components and Objects*. Springer, 2006.
- [18] L. De Moura and N. Björner, “Z3: an efficient SMT solver,” in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008.
- [19] J. C. Blanchette, L. Bulwahn, and T. Nipkow, “Automatic proof and disproof in Isabelle/HOL,” in *FroCoS*, ser. LNCS, vol. 6989. Springer, 2011, pp. 12–27.
- [20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
- [21] S. Foster and J. Woodcock, “A Dwarf Signal in CML,” COMPASS Whitepaper WP04, Tech. Rep., September 2013.
- [22] J. Holt, “Model-based Requirements Engineering for System of Systems,” in *Proc. 7th Intl. Conference on Systems of Systems Engineering (SoSE)*. IEEE, July 2012.
- [23] J. Bryans, J. Fitzgerald, R. Payne, and K. Kristensen, “SysML Contracts for Systems of Systems,” June 2014, to appear in IEEE SoSE 2014.
- [24] M. Clavel, F. Duán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, “Maude: specification and programming in rewriting logic,” *Theor. Comp. Sci.*, vol. 285, no. 2, pp. 187–243, 2002.
- [25] A. Griesmayer, Z. Liu, C. Morisset, and S. Wang, “A framework for automated and certified refinement steps,” *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 3–16, 2013.